2122

MR2039-159

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: Vladimir I. Miloushev, et al. :

Serial No.: 09/780,452 : Art Unit #2122

Filed : 9 February 2001 : Examiner:

Title : DYNAMIC CONTAINER FOR SOFTWARE PARTS AND AND METHODS OF USE :

PRELIMINARY AMENDMENT

Box Non-Fee Amendment
Honorable Commissioner for Patents
Washington, D.C. 20231

Sir:

Applicants, by the undersigned attorney, wish to amend the above-referenced Application in order to make corrections thereto.

IN THE DRAWINGS:

Please replace the Drawings originally filed of FIGS. 1-150 with the Substitute Formal Drawings of FIGS. 1-150 (154 sheets) submitted herewith.

IN THE SPECIFICATION:

Please add/replace pages 200, 222, 223, 243, 244, 245, 246, 247, 248, 402, 452, 465, 485, 491, 493, 494, 497, 501, 725 and 726 with the replacement Specification pages attached hereto.

MR2039-159

REMARKS

This case was filed concurrently with several other Divisional Applications on 9 February 2001. It has come to Applicants' attention that due to a clerical error, a number of pages were missing from each of several of the Divisional Applications. Replacement pages corresponding to those pages that were copied from the U.S. Patent and Trademark Office File Record of the Parent case, with the exception of five pages, page numbers 402, 491, 493, 494 and 501 obtained from the Attorney's File, but for which the subject matter thereof was fully disclosed in the Provisional Application for which priority was claimed in the Parent Application, have been obtained for submission in those cases. Since it is highly likely that one or more of the same pages were omitted from the subject Patent Application, Applicants are hereby submitting the attached Specification pages for incorporation in the subject Patent Application. As the subject matter of the attached pages was disclosed in the Parent Patent Application, Serial No. 09/640,898, filed 16 August 2000, and/or the Provisional Application for which priority was claimed, the addition of these pages to the subject Patent Application does not introduce any new matter.

MR2039-159

The undersigned attorney's Associate Power was filed on 8 June 2001, and a copy thereof is attached hereto. Correspondence should be continued to be sent to Doyle B. Johnson of CROSBY, HEAFEY, ROACH & MAY.

Respectfully submitted,
For: ROSENBERG, KLEIN & LEE

David I. Klein
Registration #33,253

Dated: 15 June 2001

Suite 101
3458 Ellicott Center Drive
Ellicott City, MD 21043
(410) 465-6678

To use DM_IFLT/DM_IFLTB to filter events by ID, they may be parameterized to use the event ID as the filter integer value. The min and max properties can be used to specify the range of the event IDs that are sent through the aux terminal (auxiliary flow). See the properties below for more information.

59.4. Special events, frames, commands or verbs

None.

59.5. Properties

Property "offset" of type "UINT32". Note: Offset of the filter integer value in the bus passed with the operation received on the in terminal (specified in bytes). The offset is specified from the beginning of the operation bus. The size of the integer value stored at this offset is expected to be 32-bits. Default is 0 (first field in operation bus).

Property "mask" of type "UINT32". Note: Bitwise mask ANDed with the integer value extracted from the operation bus. DM_IFLT/DM_IFLTB masks the extracted integer value before comparing it to min and max. Default is 0xFFFFFFFF (no change).

Property "min" of type "UINT32". Note: Lower boundary of the auxiliary operations. This is the lowest integer value (inclusive) that is considered auxiliary. If filtering events, this is the lowest event ID that is considered auxiliary. Default is 0.

Property "max" of type "UINT32". Note: Upper boundary of the auxiliary operations. This is the upper most integer value (inclusive) that is considered auxiliary. If filtering events, this is the upper most event ID that is considered auxiliary. Default is 0xFFFFFFFF.

60. Encapsulated interactions

None.

61. Specification

62. Responsibilities

12. If the operation filter integer value received on the in terminal is between min and max, pass operation through the aux terminal (auxiliary flow).

90. Theory of operation

90.1. Mechanisms

Idle generation

Idle generation becomes enabled or disabled when DM_IEV receives EV_REQ_ENABLE or EV_REQ_DISABLE respectively (through the idle terminal). By default, idle generation is enabled.

The idle generator is a tight loop that will continuously generate EV_IDLE events through the idle terminal. The generation will stop if the event return status is CMST_NO_ACTION or CMST_BUSY or if EV_REQ_DISABLE is received on the idle terminal.

Passing the external event

The incoming event is passed through the out terminal either before or after the idle generation. This is determined by the value of the idle_first property. If the property is TRUE, the incoming event is sent out after the idle generation, otherwise its sent before.

90.2. Use Cases

Idle generation after passing the event through

1. The counter terminal of in sends an event to DM_IEV. The idle_first property is FALSE.
2. The event is passed through the out terminal.
3. If the idle generation is enabled, EV_IDLE events are continuously generated and sent out through the idle terminal. The idle generation stops either when an EV_REQ_DISABLE event is received through the idle terminal or an event status of CMST_NO_ACTION or CMST_BUSY is returned.
4. DM_IEV returns with the status obtained in step 2 above.

Idle generation before passing the event through

5. The counter terminal of in sends an event to DM_IEV. The idle_first property is TRUE.

6. If the idle generation is enabled, EV_IDLE events are continuously generated and sent out through the idle terminal. The idle generation stops either when an EV_REQ_DISABLE event is received through the idle terminal or when an event status of CMST_NO_ACTION or CMST_BUSY is returned.
7. The event is passed through the out terminal.

Notes

DM_IEV is an idle feed generator driven by external events. Whenever it receives an incoming call (event), the idle generator propagates it to its output and then starts generating idle feed, or pulse event, (EV_IDLE) through its idle terminal. When it receives indication that there is no more need for idle feed, it returns to the original caller.

Together with DM_DWI, this part forms a complete implementation of the *run to completion* pattern. Whenever an incoming call is received, DM_IEV sends it out for processing; during this processing, one or more events may get enqueued on the desynchronizer's queue for later processing. When DM_IEV receives control back, it starts feeding events into the desynchronizer, causing all pending events to be distributed. As a result, before DM_IEV returns to its caller, all events that were generated during the processing of the original call, are completely served. In conjunction with the poly-to-drain and drain-to-poly adapters, this mechanism can provide run to completion for practically any input interface.

Terminators

DM_STP, DM_BST, DM_PST, DM_PBS – Event and Operation Stoppers

Fig. 66 illustrates the boundary of the inventive DM_STP part.

Fig. 67 illustrates the boundary of the inventive DM_BST part.

Fig. 68 illustrates the boundary of the inventive DM_PST part.

Fig. 69 illustrates the boundary of the inventive DM_PBS part.

the con terminal. If FALSE, DM_CTR will not output anything. It will just pass the operation call through the out terminal. Default is TRUE.

Property "op1-op16" of type "ASCIIZ". Note: These properties are the names of the first 16 operations. DM_CTR uses these names to identify the operation call in the call information output. If the operation name is empty, the operation ID is used. Default is "".

7. Encapsulated interactions

None.

8. Specification

9. Responsibilities

1. Dump the call information to either the debug console or send an EV_MESSAGE event containing the output.
2. Pass all operation calls on the in terminal out through the out terminal.

10. Theory of operation

10.1. State machine

None.

10.2. Main data structures

None.

10.3. Mechanisms

Dumping the call information

DM_CTR will assemble all output into one buffer and then dump the entire buffer either to the debug console or by sending an EV_MESSAGE event through the con terminal.

DM_CTR determines where to send the output by checking if the con terminal is connected on activation. If con is connected, DM_CTR will send EV_MESSAGE events that contain the output. This enables the output to be sent to a different medium other than the debug console (i.e. serial port). If con is not connected, the output will always go to the debug console.

The format of the call information before DM_CTR passes the incoming call through out is:

<instance name> [#<instance id>] (<re-enterance call #>) <operation name/id> (<operation call #>) called\n

The format of the call information after DM_CTR passes the incoming call through out is:

<instance name> [#<instance id>] (<re-enterance call #>) <operation name/id> (<operation call #>) returned <status text> [<status code>]\n

Example:

MyCTRDump [#3451879] (1) 'MyOpName' (3) called\n
MyCTRDump [#3451879] (2) 'MyOpName' (4) called\n
MyCTRDump [#3451879] (2) 'MyOpName' (4) returned CMST_OK [0]\n
MyCTRDump [#3451879] (1) 'MyOpName' (3) returned CMST_OK [0]\n

In the example above, 'MyOpName' was called a total of 4 times.

Field	Description
instance name	Unique name of DM_CTR supplied by user (name property).
instance id	Unique instance id of DM_CTR (assembled by DM_CTR).
re-enterance call #	Value that uniquely identifies the operation call in case of recursive calls to operations through the same interface. This makes it easy to trace recursive operation calls.
operation call #	Value that indicates the number of times operations have been called through this interface. DM_CTR only keeps track of the first 16 operations.

Field	Description
operation name	Name of operation invoked. If the operation does not have a name, DM_CTR will output the following "operation #XX" where XX is the operation number.
status text	Return status (text form) of operation invoked through DM_CTR's out terminal.
status code	Return status code of operation invoked through DM_CTR's out terminal.

10.4. Use Cases

Tracing/debugging the program flow through connections (output sent to the debug console)

1. Insert DM_CTR between part A and part B. Part A's output terminal is connected to DM_CTR's in terminal and Part B's input terminal is connected to DM_CTR's out terminal.
2. Parameterize DM_CTR with an instance name and operation names (instance and operation names are optional).
3. Activate DM_CTR.
4. As Part A invokes operations through its output terminal connected to DM_CTR, the operation calls come to DM_CTR's in terminal. DM_CTR displays the call information to the debug console.
5. The operation call is passed out through DM_CTR's out terminal and the operation on part B's input terminal is invoked. The return status from the operation call is returned to the caller.

Tracing/debugging the program flow through connections (output sent to other mediums)

1. Insert DM_CTR between part A and part B. Part A's output terminal is connected to DM_CTR's in terminal and Part B's input terminal is connected to DM_CTR's out terminal.
2. Connect DM_CTR's con terminal to Part C's in terminal.
3. Parameterize DM_CTR with an instance name and operation names (instance and operation names are optional).
4. Activate DM_CTR.
5. As Part A invokes operations through its output terminal connected to DM_CTR, the operation calls come to DM_CTR's in terminal. DM_CTR sends an EV_MESSAGE event containing the call information through the con terminal.
6. Part C receives the EV_MESSAGE event and sends the call information out a serial port to another computer.
7. The operation call is passed out through DM_CTR's out terminal and the operation on part B's input terminal is invoked. The return status from the operation call is returned to the caller.

DM_BSD – Bus Dumper

Fig. 80 illustrates the boundary of the inventive DM_BSD part.

DM_BSD is used to trace the program execution through part connections. DM_BSD can be inserted between any two parts that have a unidirectional connection.

When an operation is invoked on its in terminal, DM_BSD dumps the operation bus fields. The dump goes to either the debug console or by sending an EV_MESSAGE event through the con terminal (if connected). The operation is then forwarded to the out terminal. When the call returns, DM_BSD dumps the bus again. The dumping of the bus before and after the operation call can be selectively disabled through properties. DM_BSD does not modify the operation bus.

In order to interpret the operation bus, DM_BSD must be parameterized with a pointer to an interface bus descriptor (bus_descp property). This descriptor specifies

the format strings and operation bus fields to be dumped. The format string syntax is the same as the one used in printf.

The order of the fields in the descriptor needs to correspond to the order of the format specifiers in the format string. The descriptor may have any number of format strings and fields. The only limitation is that the total size of the formatted output cannot exceed 512 bytes. Please see the reference of your C or C++ run-time library for a description of the format string specifiers.

DM_BSD's output can be disabled through properties. When disabled, all operations are directly passed through out, allowing for selective tracing through a system. By default, DM_BSD will always dump the operation bus according to its descriptor.

Each DM_BSD instance is uniquely identified. Before dumping the operation bus, DM_BSD will identify itself. This identification includes the DM_BSD unique instance id, recurse count of the operation invoked and other useful information. This identification may also include the value of the name property.

Note As both terminals of DM_BSD are of type I_POLY, care should be taken to use only compatible terminals; DM_BSD may not always check that the contract ID is the same.

---

11. Boundary

11.1. Terminals

Terminal "in" with direction "In" and contract I_POLY. Note: v-table, infinite cardinality, floating, synchronous. All operations invoked through this terminal are passed through the out terminal. DM_BSD does not modify the bus passed with the operation.

Terminal "out" with direction "Out" and contract I_POLY. Note: v-table, cardinality 1, floating, synchronous. All operations invoked on the in terminal are passed through this terminal. If this terminal is not connected, DM_BSD will return with CMST_NOT_CONNECTED after dumping the bus information. DM_BSD does not modify the bus passed with the operation.

Terminal "con" with direction "Out" and contract I_DRAIN. Note: v-table, cardinality 1, floating, synchronous. If connected, DM_BSD sends an EV_MESSAGE event containing the bus dump through this terminal. In this case no debug output is printed.

11.2. Events and notifications

Outgoing Event	Bus	Notes
EV_MESSAGE	B_EV_MSG	DM_BSD sends an EV_MESSAGE event containing the bus dump through the con terminal (if connected). This allows the dump to be sent to mediums other than the debug console.

11.3. Special events, frames, commands or verbs

None.

11.4. Properties

Property "name" of type "ASCIIZ". Note: This is the instance name of DM_BSD. It is the first field printed before the bus dump. If the name is "", the instance name printed is "DM_BSD". Default is "".

Property "enabled" of type "UINT32". Note: If TRUE, DM_BSD will dump the call information to either the debug console or as an EV_MESSAGE event sent through the con terminal. If FALSE, DM_BSD will not output anything. It will just pass the operation call through the out terminal. Default is TRUE.

Property "bus_descp" of type "UINT32". Note: This is the pointer to the operation bus descriptor used by DM_BSD. It describes the output format and the operation bus fields. This property must be set and contain a valid descriptor pointer. This property is mandatory.

details on the virtual entity container, see Appendix 6. VECON – Virtual Entity Container and Appendix 13. Interfaces Used by Described Mechanisms.

9.2. VPROP – Virtual Property Helper

The virtual property helper is used to maintain data associated with a single instance of a virtual property. It uses the following structure to keep said data.

```
typedef struct VPROP
  {
  char   *namep;  // name of the property
  uint16  type;   // property data type
  void   *valp;   // pointer to value
  uint32  len;    // length of the value
  CM_OID  oid;    // object to allocate on behalf of
  } VPROP;
```

The name of the property is kept by reference; the helper is responsible to allocate the storage. The same is valid for the value of the property. The name/value storage allocation happens at the same time when the virtual property is added (created) and therefore has the same life scope as the property itself.

The reason for this storage being allocated dynamically is that there is no explicit limit on the length of the property name. The same is valid for the property value.

The set of virtual properties is maintained by an instance of the VECON virtual property container.

For more details on the virtual property helper, see Appendix 6. VECON – Virtual Property Container and Appendix 13. Interfaces Used by Described Mechanisms.

9.3. VPDST – Virtual Property Distributor

The virtual property distributor is used to distribute the value of a virtual property to the current set of contained elements, when the array receives a request to set said virtual property (note that this request is typically received through the component boundary, not through the prop terminal).

16.2. Mechanisms

Driver initialization and cleanup

When the VxD containing DM_VXFAC is loaded (or is opened using CreateFile()), DM_VXFAC receives a EV_VXD_INIT event. In response to this event, DM_VXFAC creates an instance of the device's class (specified by the class_name property). DM_VXFAC then parameterizes and activates the instance. DM_VXFAC enforces that only one instance of the driver's class may exist at any time - DM_VXFAC fails additional EV_VXD_INIT events.

When the VxD is unloaded (or is closed using CloseHandle() or DeleteFile()), DM_VXFAC receives an EV_VXD_CLEANUP event. In response to this event, DM_VXFAC deactivates and destroys the device instance. Additional EV_VXD_CLEANUP events are ignored.

Dispatching open/close operations to device instances

When the device is opened using the CreateFile() Win32 API, DM_VXFAC receives a DIOC_OPEN message (through the EV_VXD_MESSAGE event). DM_VXFAC fills out a B_DIO bus and translates this message into a dio.open operation.

When the device is closed using the CloseHandle() Win32 API, DM_VXFAC receives a DIOC_CLOSEHANDLE message (through the EV_VXD_MESSAGE event). DM_VXFAC fills out a B_DIO bus and translates this message into dio.cleanup and dio.close operations.

If the dio.open, dio.cleanup or dio.close operations complete asynchronously (return CMST_PENDING), DM_VXFAC waits on a semaphore until the operation completes. When dio.complete is called to complete the pending operation, the semaphore is signaled and DM_VXFAC completes the operation. This is necessary because the open and close operations issued by the operating system must complete synchronously.

Dispatching I/O control operations to device instances

I/O control operations are sent as EV_VXD_MESSAGE events (W32_DEVICEIOCONTROL message) when an application uses the DeviceIOControl()

Property "vendor_id" of type "uint32". Note: Vendor ID.

Property "device_id" of type "uint32". Note: Device ID.

Property "subsys_vendor_id" of type "uint32". Note: Subsystem Vendor ID

Property "subsys_device_id" of type "uint32". Note: Subsystem Device ID

Property "reg_root" of type " unicodez". Note: registry path to the specified device instance key (per device instance)

Property "class_name" of type "asciiz". Note: class name of part to be created for handling this device instance

Property "device_name" of type "unicodez". Note: name to use for registering the device

Property "friendly_name" of type "unicodez". Note: Win32 alias (does not include the \??\ prefix)

Property "port_base" of type "BINARY (uint64)". Note: I/O port base. (8-byte physical address). Could be more than 1 per device.

Property "port_length" of type "uint32". Note: Specifies the range of the I/O port base. Could be more than 1 per device.

Property "mem_base" of type "BINARY (uint64)". Note: The physical and bus-relative memory base (8-byte physical address). Could be more than 1 per device.

Property "mem_length" of type "uint32". Note: Specifies the range of the memory base.. Could be more than 1 per device.

Property "irq_level" of type "uint32". Note: Bus-relative IRQL. Could be more than 1 per device.

Property "irq_vector" of type "uint32". Note: Bus-relative vector. Could be more than 1 per device.

Property "irq_affinity" of type "uint32". Note: Bus-relative affinity. Could be more than 1 per device.

Property "dma_channel" of type "uint32". Note: DMA channel number. Could be more than 1 per device.

Property "dma_port" of type "uint32". Note: MCA-type DMA port. Could be more than 1 per device.

8. DM_CBFAC binds to the existing instance and increments the construction reference count by one. DM_CBFAC passes the instance id back to MyPart.

9. MyPart activates the singleton through fact.activate passing the instance id returned from fact.create. Since the singleton is already active, DM_CBFAC increments the activation reference count and returns.

10. Steps 7-9 may be repeated several times.

11. Eventually MyPart needs to deactivate and destroy the instances created in the steps above. MyPart calls fact.deactivate and fact.destroy for each instance created in the steps above.

12. DM_CBFAC decrements the activation and construction reference counts by one on each call to fact.deactivate and fact.destroy respectively. As soon as the reference counts reach zero, the factory deactivates and destroys the singleton.

Enforcing one-time part instantiation (singletons) using specified part class in B_A_FACT bus

This use case is exactly the same as the one described above except the singleton part class name is specified in the B_A_FACT bus. This may be used when the name of the singleton part class is known only at run-time (i.e., read from registry, etc.)

The steps are repeated below for clarity:

1. The structure in the above diagram is created and connected.

2. DM_CBFAC is parameterized with the following:

   a. force_dflt_class = FALSE

3. The structure in the above diagram is activated.

4. Some time later, MyPart needs to create a singleton part. MyPart invokes fact.create specifying the part class name in B_A_FACT.namep.

5. DM_CBFAC tries to bind to an existing instance using the instance name specified in the bus. The binding fails so DM_CBFAC creates a new

end of the event (-1 specifies the last byte). Note that the context storage must be at least sizeof (_ctx) big. The default value is (-sizeof (_ctx)) (end of the event)

6.3. Events and notifications

Terminal: ctl

ZP_E2FAC does not define the set of events or the structure of the event bus. The event bus for the following events must at a minimum contain storage for the part instance ID. The event IDs are specified as properties.

Incoming Event	Dir	Bus	Notes
(create_ev)	in	any	ZP_E2FAC creates a part instance out its fac terminal. The event bus must contain also part class name or a reference to a part class name.
(destroy_ev)	in	any	ZP_E2FAC destroys the part instance out its fac terminal.
(activate_ev)	in	any	ZP_E2FAC activates the specified part instance out its fac terminal.
(deactivate_ev)	in	any	ZP_E2FAC deactivates the specified part instance out its fac terminal.
(enum_get_first_ev)	in	any	Gets the first part instance from a part instance holder. The event bus must contain storage for the enumeration context. The size of the enumeration context is sizeof (_ctx).
(enum_get_next_ev)	in	any	Gets the next part instance from a part instance holder. The event bus must contain storage for the enumeration context. The size of the enumeration context is sizeof (_ctx).

7. Environmental Dependencies

None.

7.1. Encapsulated interactions

None.

7.2. Other environmental dependencies

None.

2. ZP_E2FAC invokes create operation out its fac terminal
3. ZP_E2FAC receives activate_ev on its ctl terminal
4. ZP_E2FAC invokes activate operation out its fac terminal
5. ZP_E2FAC receives deactivate_ev on its ctl terminal
6. ZP_E2FAC invokes deactivate operation out its fac terminal
7. ZP_E2FAC receives destroy_ev on its ctl terminal.
8. ZP_E2FAC invokes destroy operation out its fac terminal.

11.2. Automatic activation and deactivation

The user of ZP_E2FAC has set the activate_ev and deactivate_ev properties to zero.

1. ZP_E2FAC receives create_ev on its ctl terminal
2. ZP_E2FAC invokes create operation out its fac terminal
3. If the part creation succeeds, ZP_E2FAC invokes activate operation out its fac terminal
4. ZP_E2FAC receives destroy_ev on its ctl terminal.
5. ZP_E2FAC invokes deactivate operation out its fac terminal
6. If the part deactivation succeeds, ZP_E2FAC invokes destroy operation out its fac terminal.

12. Notes

The byte order in the ID matches the default byte order supported by the CPU.

Appendix 1 – Interfaces

This appendix describes preferred definition of interfaces used by parts described herein.

I_DRAIN – Event Drain

Overview

The Event Drain interface is used for event transportation and channeling. The events are carried with event ID, size, attributes and any event-specific data. Implementers of this interface usually need to perform a dispatch on the event ID (if they care).

Events are the most flexible way of communication between parts; their usage is highly justified in many cases, especially in weak interactions. Examples of usage include notification distribution, remote execution of services, etc.

Events can be classified in three groups: requests, notifications and general-purpose events. The events sent through this interface can be distributed synchronously or asynchronously. This is indicated by two bits in the attr member of the bus.

Additional attributes specified within the same member indicate whether the data is constant (that is, no recipient is supposed to modify the contents), or whether the ownership of the memory is transferred with the event (self-ownership). For detailed description of all attributes, see the next section.

There are two categories of parts that implement I_DRAIN: transporters and consumers. Transporters are parts that deliver events for other parts, without interpreting any data except id and, possibly, sz. They may duplicate the event, desynchronize it, marshal it, etc.

In contrast, consumers expect specific events, process them by taking appropriate actions and using any event-specific data that arrives with the event. In this case the event is effectively "consumed".

If the event is self-owned, consumers need to release it after they are done processing. This is necessary, as there will be no other recipient that will receive the same event instance after the consumer. Transporters do not need to do that, they

```
END_EVENTX

MY_EVENT *eventp;
cmstat    status;

/* create a new event */
status = evt_alloc (MY_EVENT, &eventp);
if (status != CMST_OK) . . .

/* set event data */
eventp->my_event_data = 128;

/* raise event through I_DRAIN output */
out (drain, raise, eventp);
```

W

Example:

```
B_ITEM itembus;
char   buffer [256];
cmstat status;

/* initialize item bus */
itembus.qry_hdl = 0;
itembus.pathp   = "customer[0].name";
itembus.stgp    = buffer;
itembus.stg_sz  = sizeof (buffer);
itembus.attr    = 0;

/* get item data for 'customer[0].name' */
status = out (item, get, &itembus);
if (status != CMST_OK) return;

/* print customers name */
printf ("The first customers name is %s\n", buffer);
```

See Also: DM_REP, I_QUERY, EV_REP_NFY_DATA_CHANGE

---

set

Description: Set an item specified by data path

In:

qry_hdl	Handle to query or 0 to use absolute path
pathp	Data path (ASCIIZ zero-terminated) If qry_hdl != 0 then data path starts from the current query position. If qry_hdl == 0 then data path starts from the root.

chk

in : id - id of part in the array

namep - null-terminated property name

type - type of the property value to check

bufp - pointer to buffer containing property value

val_len - size in bytes of property value

out: void

act: check if a property can be set to the specified value

s : CMST_OK - successful

CMST_NOT_FOUND - the property could not be found or the id is invalid

CMST_REFUSE - the property type is incorrect or the property cannot be changed while the part is in an active state

CMST_OUT_OF_RANGE - the property value is not within the range of allowed values for this property

CMST_BAD_ACCESS - there has been an attempt to set a read-only property

CMST_OVERFLOW - the property value is too large

CMST_NULL_PTR - the property name pointer is NULL or an attempt was made to set default value for a property that does not have a default value

- redirect to Part Array API

get_info

in : id - id of part in the array

namep - null-terminated property name

out: type - type of property [CMPRP_T_XXX]

attr - property attributes [CMPRP_A_XXX]

act: retrieve the type and attributes of the specified property

s : CMST_OK - successful

CMST_NOT_FOUND - the property could not be found

or the id is invalid

- retrieve element oid
- redirect to ClassMagic API